AN INSIGHT TO NASCOM ROM BASIC

What follows is the result of many hours delving into the intricasies of Nascom's ROM Basic, as far as I can tell the information here is correct but no guarantees are given. The best way to study the ROM is to re-organise memory so that the ROM occupies say C000 DFFF and then copy it to RAM at E000 FFFF. In this way it is possible to break and single step whilst running in RAM, with a dissassembly listing in one hand and these notes in the other pointing you in the right direction. NASSYS owners wishing to use NMI facilities must return to the monitor after initialising BASIC to reset $NMI, and NASSYS users should return to Basic by EFFFD as the Z command doesn't set the breakpoint.

Let's start with a few points omitted from Nascom's manual.

The RESTORE command may be followed by a line number to enable DATA reading to commence from any line desired.

The CLEAR command may be followed by 2 arguments, CLEAR(X,Y), where X = string space and Y = user defined top of RAM. In this way it is possible to reserve RAM without re-initialising BASIC.

The WIDTH function, when set to 255, will not generate a NEWLINE at all.

Early copies of NASCOM's notes omitted to mention an MO Error meaning MISSING OPERAND which occurs if an "=" is not followed by a correct statement.

It is possible to stop and then single step a running program by repeatedly entering CTRL/ESC.

The BACKSPACE character is not properly catered for. During INPUT (or during normal line entry under T4/B-BUG) a BACKSPACE decrements the "number of characters entered" counter in B, but doesn't decrement the WIDTH counter in 10AB, suprious NEWLINEs can therefore occur. The WIDTH counter is also checked during the PRINT command and if BACKSPACEs have been output again spurious NEWLINEs can occur and the TAB function is upset. I think that this is because the output routines are designed to work to a hard copy device where true backspaces are not possible, if the WIDTH is set to 255 (where it should be) spurious NEWLINEs will not occur.

A semi-colon need not be entered except at the end of a line e.g. PRINT SPC(8);"YOU HAVE";X;"POUNDS"; may be entered PRINT SPC(8)"YOU HAVE"X"POUNDS";

POINT SET and RESET are called reflectively through the workspace and user routines can be written.

Nascom's ROM Basic occupies E000 FFFFH it uses 1000 10F9H for a scratchpad and 10FAH upwards as program space, simple string and numeric variable data is stored above the program, and array variables above that. RAM top is set on entry via the 'Memory size?' prompt, or by default to the end of RAM which BASIC finds for it's self. The stack top is set on entry 50 bytes below RAM top, giving 50 bytes of string space above the stack.

Progam entry commences at 10FAH, the first two bytes of each line form a link address pointing to the start of the next Basic line, if this address is 0000 then this indicates the top of the program and the address of the following byte is stored at 10D6H. The second and third bytes hold the Basic line number as a sixteen bit hex number. The Basic line then follows stored as it was entered except that a reserved word is stored as a single hex byte in the range 80H to CFH (see appendix A) unless it is found after a REM or within quotes. The line is terminated by a single 00.

For example

```
10 PRINT"HELLO
(The space after the line number is inserted by LIST)

10FA 06 11                  link address pointer
10FC 0A 00                  line number 10
10FE 9E                     PRINT
10FF 22                       "
1100 48 45 4C 4C 4F         HELLO
1105 00                     terminator
1106 START OF NEXT LINE
```

Above the program area resides function and simple string and numeric variable data. Each numeric and string variable takes up six bytes (more on function variables later), the first two identify the variable, a string variable being denoted by bit 7 of the first byte being set e.g.

```
Variable ABV    42 41
         C      00 43      note only 2 characters and
         AB$    C2 41        saved in reverse order
```

The next four bytes form an accumulator to define the variable. With a string the first byte holds the string length, the second is not used, and the third and fourth hold the address of the start of where the string is to be found (I suppose this is not really an accumulator but it serves as a useful common name with the numeric 4 bytes and quite a few routines are comon to both). A typical string variable looks like this:-

```
80 41 04 xx 08 9F
```

This would mean that string A$ is 4 characters long, and is to be found at 9F08. With a numeric variable the 4 bytes form a mantissa and exponent in 'normalized floating point binary notation' (PHEW!!). As you should already understand a number can be expressed as a value multiplied by an exponent, for instance a decimal number 146 can be shown as :-

```
   1.46 E+2 =1.46*10^2
or 14.6 E+1 =14.6*10^1
   1460 E-1 =1460*10^-1(1/10)
```

The exponent E+n tells you to shift the decimal point n places to the left if a minus value, or right if a positive value. Exactly the same thing can be done with binary numbers, for example the number 1001B (9decimal) can be shown thus :-

```
    1001 E+0 =1001*2^0 (1)
or 1.001 E+3 =1.001*2^3
   .1001 E+4 =.1001*2^4
```

The exponent is now telling you to shift the binary point. It is obviously no good however to have a number being expressed in different ways, so a standard format is used (normalization). This is done by always putting the binary point one place to the left of the most significant 1 ,and letting the exponent dictate the true value of the number. So:-

```
   0.1   E+1 =1          1D
   0.1   E+0 =0.1      0.5D
   0.11  E+2 =11         3D
   0.11  E+3 =110        6D
   0.101 E+5 =10100     20D
```

As can be seen, except when a value of 0 is indicated, the first digit to the right of the binary point is always a 1, and can therefore be assumed and is not stored in the mantissa, a 0 is indicated by other means (see later). And now to how the value is stored within the four bytes of the variable's accumulator. To see this best we look at the bytes in reverse order. Byte 4 holds the binary exponent as 80H+ the exponent e.g 86H = E+6 7FH = E-1 etc. An exponent value 00 holds special significance as it indicates that the variable is 0 regardless of the contents of the mantissa. Bytes 3,2,1 in that order then hold the mantissa with bit 7 of byte 3 indicating the sign (set -ve reset +ve) E.g.

00 43 00 00 0C 86

This would show that variable C would be 35D, a breakdown follows:- 35D=100011B E+0 normalizing gives 0.100011 E+6 the most significant 1 is then stripped and the variable made up as follows.

```
BYTE     4           3           2           1
        86          0C          00          00
     EXPONENT    MANTISSA    MANTISSA    MANTISSA
     10000110   0 0001100   00000000    00000000
                 |
        E+6   Sign   00011
              +ve
```

A function variable takes up 6 bytes the first two identify the variable, bit 7 of the second byte is set to identify a function variable. The third and fourth bytes contain an address pointer pointing to the expression defining the function in the Basic program, the next 2 bytes identify the dummy variable used in the definition. For example:-

```
110 DEF FN AB (CD)=CD+2
would be stored as
42 C1      xx xx        44 43

function   address      dummy
ident      following    variable
 AB        the = in       CD
           the Basic
             line
```

The address of the byte after the last simple variable is stored at 10D8 and this indicates the end of the simple variables and the start of the array variables. The first two bytes of an array identify the array variable in exactly the same way as a simple variable, the third and fourth bytes hold the hex length of the array data following, the fifth byte holds the number of dimensions and then subsequent pairs of bytes hold hex number of elements within each dimension, the dimension sizes being stored in reverse order to which they were entered by the DIM command. Each element is then specified in turn (the order is shown in Nascom's Basic manual), each element using a 4 byte accumulator in exactly the same as simple variables. The address of the byte after the end of all array variables is stored at 10DA.

Actual string data is stored in a couple of ways, with a simple string e.g 100 A$="THIS IS A STRING" the string address pointer in the variable data will point directly to the string in line 100 in the Basic program, but with a more complex string e.g 110 B$=A$+CHR$(18) the string has to be constructed and the string area above the stack will be used. Basic uses the string area in what

appears at first sight to be a wasteful manner, if a string variable whose string is already occupying part of the string area is re-assigned a new value the new value will be added to the bottom of the string area and the old will be left behind taking up space, this is because the garbage collection of these dead strings takes a long time, the time being a function of the number of live strings and in the case of large arrays can run into minutes, so garbage collection only takes place when absolutely neccessary i.e when string space runs out.

That leaves just GOSUB and FOR/NEXT loops which require storage space, these in some respects are similar in that they both need to refer back (when encountering a RETURN or NEXT), in fact a common sub-routine at E356 is integral to operation of both. Reference back information is stored on the stack, progressively lower, as each one is encountered. The Z80 stack operation, of course, also works on the same last in first out mode. And it is because of this LIFO operation that a FOR/NEXT will only remain valid within its own level i.e. on execution of a GOSUB previously set FOR/NEXT loops will remain frozen above the GOSUB data on the stack, similarily any FOR/NEXT set up during a GOSUB routine will be cancelled when a RETURN is executed. Data is stored in the following formats :-

GOSUB in 5 bytes

8C	xx xx	yy yy
GOSUB ident.	hex line number of calling line	address of byte following GOSUB command in calling line

FOR/NEXT in 16 bytes

81	zz zz	aa	ss ss ss ss	ee ee ee ee	xx xx	yy yy
FOR ident.	Address of start of variable	Set 0 if step is 0	Step accumulator	End accumulator	Hex line number of calling line	Address of byte at end of FOR statement

The Basic program is basically in 8080 code, there are only a few Z80 only instructions and these are found in the NASCOM interface routines. There is also a programming technique which needs explaining before we continue as it can cause perplexion. Look for example at:-

```
E3AD 1E 02      LD E 02
E3AF 01 1E 14   LD BC 141E
E3B2 01 1E 00   LD BC 001E
E3B5 01 1E 12   LD BC 121E
```

Entry to this part of the program is in fact always at a 1E byte the LD E n being the important thing the LD BC nn instructions being there to 'cover up' the LD E n instructions that aren't required. In the example above the appropriate entry point will load E with a value which is used to add into the list of error codes.

For those dissassembling the ROM the following areas contain data and not executed code.

E008 E011 call address to obtain argument from a USR call / call address to return value to Basic from a user routine/ JP E73C executing this sets the NMI vector to FEDE. (not called)

E0B7 E10E messages/ Bytes free/ Nascom ROM Basic/ Memory size

E10F E142 addresses of reserved word routines SGN MID$

E143 E259 reserved word table, bit 7 of the first letter of each word is set 80H terminates the table

E25A E2A3 more addresses of reserved word routines END NEW

E2A4 E2B8 3 byte vector address for maths operators + to OR, the first byte is to give the operator's evaluation priority

E2B9 E2DE list of errors

E2DF E33F copied to 1000 to set up the scratchpad

E340 E355 messages/ Error in/Ok /Break

EBD9 EBEB message/Re-do from start

ECC1 ECD1 message/ Extra ignored

F58E F5A2 messages/ File found/ Bad

F6B6 F6C6 data for maths routines

FA91 FAA4 decimal data, 1st 4 bytes hold standard accumulator of 0.1D, then in 3 byte hex blocks 100,000 10,000 1,000 100 10 1

FB3A FB5A data for maths routines

FBFA FBFC unused

FC4A FC66 data for maths routines

FCA3 FCC7 "

The only other problem is that the byte following a CD 90 E6 (CALL E690) is a data byte used by the sub-routine.

Let us now commence with the scratchpad 1000 10F9

1000 C3 AE E0 JP E0AE this is not called by Basic but if you were to enter Basic via E1000 you would enter by a normal warm start except that program area and variable area would be initialised too.

1003 C3 nnnn vector for USR call

```
1006 D3 nn    IN A (n)   subroutine set up and then called
1008 C9       RET        by IN and WAIT commands

1009 D6 n1    SUB n1     mathematical subroutine values
100B 6F       LD L A     n1-n4 are entered by the calling
100C 7C       LD A H     routines before it's execution
100D DE n2    SBC n2
100F 67       LD H A
1010 78       LD A B
1011 DE n3    SBC n3
1013 47       LD B A
1014 3E n4    LD A n4
1016 C9       RET
```

1017 103D RND workspace

103E DB n OUT (n) A set up and called by OUT
1040 C9 RET

1041 single byte holding NULL value

1042 single byte holding WIDTH value

1043 single byte used for calculating number of 14 space blocks

1044 ???

1045 if set non-zero output to CRT etc will cease until reset (INPUT command will reset)

1046 1047 16 bit counter used to count lines during LIST

1048 1049 16 bit byte holding LINES value

104A 104B 16 bit checksum store, for CSAVE*/CLOAD*

104C setting this to a non-zero value will cause a crash when a program is listed

104D set by NMI routine at FEDE and causes a Break on completion of current Basic statement

104E C3 nnnn JP nnnn vector to get entry for INPUT
1051 C3 nnnn JP nnnn vector for POINT
1054 C3 nnnn JP nnnn vector for SET
1057 C3 nnnn JP nnnn vector for RESET

105A 105B stack top

105C 105D holds hex line number (current line) to be printed by error routine, set to FFFE on entry to force return to 'Memory size' prompt, set to FFFF in direct mode so that line number not printed if error occurs, also used to check for illegal direct entry.

105E 105F holds address of start of Basic program (10FA normally)

1060 10AA 72 character line buffer with T4/BBUG entries are made directly, with NASSYS during command mode entry to buffer is only made after NEWLINE

10AB WIDTH counter, also used by TAB

10AC set non-zero signifies a DIM in progress

10AD type flag 00 means numeric 01 means string

10AE DATA flag set non-zero to indicate DATA info so that reserved words within string data are not converted to reserved word single bytes

10AF 10B0 address of RAM top

10B1 10B2 address of current position in 10B3 string buffer

10B3 10C2 4 byte string accumulators used during string manipulation

10C3 10C4 address of current bottom of strings in string area

10C5 10C6 address of position in an expression during evaluation 10C7 10C8 holds the address of the end of a FOR statement during the setting up of a FOR loop

10C9 10CA holds hex line number of current DATA position

10CB identifies various functions to the variable find routine, normally zero, it is set to 64H by FOR to stop an array variable being used by a FOR loop, CSAVE*/CLOAD* sets it to 01 so that an array variable is looked for, set to 80H by DEF to stop the (in the DEF FN n (m) statement from causing an array function variable from being set up

10CC holds last character entered into the line buffer 1060

10CD READ flag if non-zero then a READ is in progress

10CE 10CF several uses, as a storage of current position, as a single byte save/load flag during CSAVE* 00 or CLOAD* FF, keeps track of position when sent back by a NEXT

10D0 10D1 stores HL (text pointer in Basic line) during variable manipulation

10D2 10D3 Break hex line number

10D4 10D5 address of Break, used by CONT

10D6 10D7 address of start of simple variables

10D8 10D9 address of start of array variables

10DA 10DB address of end of array variables

10DC 10DD address of current position in DATA

10DE 10E3 6 byte holding area for a variable under construction, usual variable format

10E4 10E7 4 byte accumulator used during number crunching or 2 byte address pointer to start of a string during string manipulation

10E8 used to store mantissa sign info during 24 bit numeric work

10E9 10F5 buffer for constructing ASCII representation of a number , terminated by a single 00 byte and printed using normal string technique

10F6 8 bit maths store

10F7 10F8 16 bit maths store

10F9 unused

10FA Basic program commences from here

And at last we get to the ROM itself. I shall run through the ROM from the bottom, but there are a couple of general routines which are used a lot and are best explained now.

FE6D FE72 checks to see which monitor is in use, returns Z if not NASSYS.

E836 E845 looks for the first non-space character on a Basic line after the location pointed to by HL, returns Z if 00 or a colon found, C set if ASCII decimal.

E977 E97E looks at next character on a line pointed to by HL and returns NC if ASCII letter

E68A E68F compares HL with DE and returns Z if the same or C if DE > HL

E690 E698 SN error routine that checks to see if the byte following the calling routine compares with the byte on a Basic line pointed to by HL

E000 E0B6 INITIALISATION

E000 E016 IX set to 0000 and routine at FEBB initialises monitor if NASSYS is being used E019 E026 copy scratchpad from E2DF to 100

E029 E033 uses E4DF to set up rest of scratchpad, then NEWLINE

E036 E043 prints Memory size and gets reply

E046 E058 if no entry BASIC finds top of RAM

E058 E06A checks that user top is valid RAM

E06D E0B4 checks for sufficient RAM, sets stack top 50 bytes below RAM top and prints greeting. Another call to E4DF sets up to RAM top, and jumps to command at E3F8

E356 E376 used by NEXT to find FOR (will not go above a GOSUB), by RETURN to by pass any FOR data to get to GOSUB, by FOR to see if FOR loop being set up is already on stack (if it is old data on stack will be re-set up allowing one to leave FOR/NEXT loops, before they have been completed, with impunity)

E379 E387 moves variable data upwards to insert new info, checks to see there is room.

E38A E3A1 check to see that there is 30H+(2*C) bytes free

E3A2 E3F3 error printing routine, value of E is used to add into error table, line number is printed if relevant

E3F6 E4B6 COMMAND MODE

E3F6 enter command mode, print Ok, and get entry line. The first entry on the line is then converted to a hex number (if there is one) and the rest of the line's reserved words converted to single bytes. If there was no number at the start a jump out to E816 is made to execute the line directly. Otherwise the line is entered/deleted

E427 E43F firstly looks for the line, if it's not there and only a line number entered then UL error, otherwise treated with one or more of following.

E443 E457 deletes the line

E45A E479 inserts the new line

E47C E496 inserts the new link address pointers and then returns to the command mode

E499 E4B6 finds a line, returns Z and NC if line not found and there is none higher, or Z and C if found . HL = START NEXT LINE

E4B9 E4FB NEW, clear variables, workspace, and program space and return to command mode

E4FC E506 print ? for INPUT and then jump to vector at 104E for reply

E509 E5C0 processes an input line and converts it, reserved words to single byte and lower case to upper except after REM or DATA or in quotes. A terminating 00 is then entered and return

E5C1 E687 entry point is E5F2 or E607. Routine loads a line of text into the 1060 buffer, if entry at E5F2 and a NASSYS monitor is being used in normal mode, entry line will be obtained at FEE8

E69B E6CB prints a character in A provided (1045)=0, checks WIDTH count and outputs a NEWLINE if required

E6CC E6DC gets keyboard entry, strips bit 7 and complements (1045) if OF entered (inhibit/allow output)

E6DD E730 LIST finds line if argument given, otherwise start at bottom, uses command table to convert single byte reserved back to full word, unfortunately does not look for quotes and converts graphics within strings too (you can't edit such lines)

E733 E73B initialises LINES counter

E73C E742 set NMI vector to FEDE. Except for being a jump vector at E00F this is not used

E745 E776 decrement LINES counter, stopping listing if count 0 and waits for continue or stop from keyboard

E779 E7F1 set up a FOR loop

E7F2 E7F6 call made from here after each statement during RUN to see if ESC pressed or if NMI has occured (104D) will have been set non zero)

E7F9 E803 on continuing check valid statement terminator

E806 E845 find next command to be executed and go to command routine with HL pointing to next non-space, and flags showing it's status (from E836 subroutine)

E846 E860 set DATA line number and address to program bottom by default or to appropriate line if one entered

E861 E89B called by LIST and RUN, Break if ESC or NMI otherwise continue

E891 E8B0 CONT, pick up old line number (if FFFF then command so CN error) and address in Basic program

E8B1 E8B8 set NULL

E8B9 E93F CSAVE*/CLOAD* data saved in format- 4 D2 bytes then dimension details then element accumulators finally 2 byte checksum

E940 E94C add in checksum

E94D E974 rest of CLOAD*/CSAVE*

E97F E9A5 ASCII string evaluating routine

E97F E988 converts string to floating point in 10E4 accumulator and ensures it is +ve

E98B E9A2 converts 10E4 accumulator to HEX in DE, ensures range + 65535 to -32768

E9A5 E9C7 convert ASCII string pointed to by HL to a hex number in DE, must be integer 0-65529, used for line numbers, this is the only routine not using floating point

E9CA EA0D CLEAR, if argument entered then move stack top down if there is room

EA10 EA19 RUN, initialise variables and then treat as GOTO

EA1C EA2C GOSUB, store current position and then treat as GOTO

EA2D EA48 GOTO, continue running from line number specified

EA4B EA6E RETURN

EA70 EA84 ignore REM and DATA

EA87 EAE0 LET, find the variable concerned, set up if not already there, calculate value and enter it

EAE1 EAFC ON, get argument value and step through line numbers to find one required

EAFF EB1C IF

EB1F EB71 PRINT

EB74 EBAC NEWLINE

EB74 EB79 executes a NEWLINE if you backspace too far during line input at E609

EB81 executes NEWLINE

EB86 resets line character counter

EB8A sends out NULLs

EB98 EBAC checks to see if a NEWLINE is needed when printing numbers in 14 character blocks

EBAF EBD6 handles SPC and TAB

EBEC EBFA error during variable load has occured, if DATA READ then SN error, if INPUT then Re-do from start

EBFD EC29 INPUT, print leading string then ? and obtain reply, then treat as READ

EC2C ECC0 READ, load variables with data

ECD2 ECF3 search for next DATA statement

ECF6 ED41 NEXT

ED44 ED4C tests for type mismatch

ED4F EF1C evaluates maths and string expressions

EE70 EE7F checks to see if + or - follows on a line pointed to by HL, HL set before character if it's not

EE80/EE81 EEA7 OR/AND

EEA8 EF05 checks operator priority during maths expression evaluation

EF08 EF1C NOT

EF1F F0CF variable finding routine. If variable is not found it will be set up, returns with DE pointing to the variable accumulator

EF28 DIM entry point

EFDF EFE9 sets string handling to print Ok (return to direct mode)

EFEA on handles arrays.

F0D0 F0F1 FRE, if string argument garbage collection takes place

F0F2 F0FB converts hex in A,B to floating point in 10E4 accumulator, LSB in B

F0FE F103 POS, loads (10AB) and returns via F0F2 to convert it to floating point

F106 F130 DEF

F133 F17A evaluates FN in maths expressions

F17B F186 checks for illegal direct

F189 F197 sets up function variable

DE = STRING ACCUMULATOR

F19A F440 STRING ROUTINES

F19A F1BE STR$

10BF = CONTENTS OF A
10C1 = POSITION OF STRING

DE = STRING ACCUMULATOR 10B5

F1BF F1CD saves string start address and length in top of 1083 buffer

F1CE F20C finds length of string pointed to by HL returns length in C, string terminated by 00 or 22H

F20F F226 print string pointed to by HL

F229 F24A tests for free space in string area

F24D F34D string manipulation

F350 F387 removes string accumulator from 10B3 buffer and removes string if it is at the bottom of the string area

F382 F390 LEN

F391 F3A1 ASC

F3A2 F3AF CHR$

F3B2 F3DF LEFT$

F3E2 F3E9 RIGHT$

F3EC F41B MID$

F41C F440 VAL

F441 F44A INP, sets up and calls 103E

F44D F450 OUT, sets up and jumps to 1006

F453 F47E WAIT, sets up and uses 103E

F481 F494 argument evaluation, used by routines such as INP, uses 16 bit E97F routine and then checks within (0 to 255), returns 8 bit in A

F495 F4B1 not called, sets up to run a Basic program (presumably intended to be in ROM) at 8000H

F4B4 vector to UART input routine

F4B7 F4C2 F4B7 outputs A through UART twice, F4BA outputs once

F4C3 F4F8 CSAVE format 3*D3, file name, data from 10D6

F4F9 F571 CLOAD

F574 F58D prints File found

F5A3 F5A7 PEEK

F5AA F5BA POKE

F5BB FCA2 MATHEMATICS ROUTINES

as a general rule number crunching takes place in 4 byte accumulators at 10E4 and in BC and DE (order BCDE byte 4321), these are referred to as the 10E4 and BCDE accumulatrs. Numbers are found in various formats, as an ASCII decimal string with/without exponent, as normalised binary floating point, and in between in varying forms of hex with or without exponent. It should also be noted that a hex number can be seen as unsigned or as signed twos complement or as just signed. The hex number FFFF would be 65535D, -1D, or -32767D respectively. Keeping track is complex so be warned and don't be suprised if you get a headache

F5BB F5C1 uses FA91 data to add/subtract to/from 10E4 mantissa

F5C4 F61A ADDITION/SUBTRACTION (10E4 and BCDE mantissas)

F5C8 complements to perform subtraction

F5CD addition

F61B F662 normalisation, shift CDE left, first in 8 bit then single bit

F665 F691 twos complement 24 bit CDE

F665 F66F increments CDE

F67E F691 complments CDE F672 F67D in the middle adds 24 bit in CDE to 24 bit in 10E4

F692 F6B3 denormalise, shift CDE right, first 8 bit then single bit

F6C7 F702 LOG

F708 F755 MULTIPLY

F756 F75A set up BCDE

F75B F7CE DIVIDE

F7D1 F7F9 MULTIPLY/DIVIDE routines

F7FC F810 multiply by 10 (hex to decimal)

F813 F821 tests 10E4 mantissa sign, Z if 00, C if -ve, NC if +ve. Returns A set FFH if -ve, 00 if 00, 01H if +ve. Entry at F81C complements sign so returns -ve +ve reversed

F822 F827 SGN

F82A F835 sets up 10E4 mantissa for normalisation

F838 F83B ABS

F83C F843 complements 10E4 mantissa's sign

F844 F878 10E4/BCDE manipulation, some used by string as well

F844 F850 puts 10E4 accumulator on stack

F851 F85E puts accumulator pointed to by HL into 10E4

F85F F86A loads accumulator pointed to by HL into BCDE

F868 F878 shift variable accumulator from DE to HL

F879 F88D re-insert leading one into mantissa, sign information stored at 10E8

F88E F8BA compares 10E4 with BCDE accumulator, returns C with A FFH if BCDE>10E4, NC with A 01H if BCDE<10E4, Z with A 00 if BCDE=10E4

F8BB F8DE converts 10E4 accumulator to 24 bit hex in CDE

F8DF F8E5 decrement BCDE

F8E6 F8FE INT

F8FF F919 checks that array arguments are within dimensioned range

F91A F9A2 converts ASCII number to binary floating point

F93E deals with exponent

F952 deals with decimal point

F977 deals with actual numerical part

F999 F9A2 puts the exponent in E

F9A5 F9A9 prints in

F9AD F9B7 converts 16 bit hex in HL to floating point in 10E4 sets the stack to return via print, after number has been converted to ASCII at F9B8 (used by line number printing for LIST)

F9B8 FA90 converts 10E4 accumulator to ASCII string in 10E9 buffer

FA82 FA90 checks to see if number is >999999 for exponent notation

FAA7 FAAB puts F83C on the stack to be used as return vector

FAAC FAB2 SQR sets up to use ^0.1 and then uses FAB5

FAB5 FB37 ^

FB8B FBF9 RND

FC00 FC47 COS, and SIN from FC06

FC67 FC79 TAN

FC7C FCA2 ATN

MAINLY NASCOM INTERFACE ROUTINES FROM HERE

FCC8 FCD4 executes MFLP and a delay (tape on before SAVE)

FCD5 vector to MFLP

FCD9 FD04 checks monitor, and alters A accordingly before output

FD05 FD37 loads A from keyboard and then alters according to monitor

Nascom II owners should note that the T2/T4/BBUG monitors use 1F 1E and 1D for CRLF CLS and BS respectively

FD40 FD55 looks to see if ESC or NMI during RUN

FD56 FD5E input character from UART

FD5F FD67 entry at FD68, output character through UART

FD70 FD8A scan keyboard looking for ESC

FD8B FD98 check monitor and execute CLS

FD9B FDA4 delay

FDA5 FDAC set (1042) for WIDTH

FDAD FDB8 set (1046) (1048) for LINES

FDBC FDC4 DEEK

FDC7 FDDD DOKE

FDDE FDE3 normal NASSYS cold start, IX set to FFFF signals Basic not to re-initialise monitor

FDE6 FE10 check mon and set (CURSOR) for SCREEN

FE11 FE38 convert screen coords to position on screen used by SCREEN, POINT, SET, RESET (checks to see it's on screen)

FE39 FE42 checks mon executes MFLP (no delay)

FE45 FE50 checks mon and calls output routine

FE53 FE6C checks mon and calls input routine

FE73 FE85 checks mon and calls WRITE

FE88 FE9F checks mon and calls READ

FEA2 FEA9 checks mon and returns to monitor

FEAA FEBA checks monitor and calls VERIFY if NASSYS

FEBB FED8 part of set up, if NASSYS NMI vector set to FEDE

FEDE FEE7 NMI comes here, sets (104D) to FF to cause Break at end of execution of current statement

FEE8 FF14 NASSYS command mode entry routine, calls INLIN and then copies CRT line to line buffer

FF15 FF3F converts SET, RESET, POINT arguments to screen coords, then to screen address

FF40 FF52 SET

FF55 FF76 RESET

FF79 FF93 POINT

FF96 FFC1 converts vertical input from SET,RESET,POINT to coords

FFC2 FFD0 converts horizontal input to coords

FFD1 FFD5 vector to MFLP

FFD6 FFDE checks load/save flag from CSAVE*/CLOAD* FFE1 FFE4 executes NEWLINE then jumps to input a line

FFE7 FFEA same again

FFED FFF3 patch for POINT

FFF4 FFF7 print A then NEWLINE

FFFA cold start vector

FFFE warm start vector

APPENDIX A

Commands with their single byte code and routine address. Note that TAB to < are treated as and when encountered, and that TAB and SPC have the leading bracket as part of the command.

80 END	E872	81 FOR	E779	82 NEXT	ECF6	83 DATA	EA70
84 INPUT	E8FD	85 DIM	EF28	86 READ	EC2C	87 LET	EA87
88 GOTO	EA2D	89 RUN	EA10	8A IF	EAFF	8B RESTORE	E846
8C GOSUB	EA1C	8D RETURN	EA4B	8E REM	EA72	8F STOP	E870
90 OUT	F44D	91 ON	EAE1	92 NULL	E8B1	93 WAIT	F453
94 DEF	F106	95 POKE	F5AA	96 DOKE	FDC7	97 SCREEN	FDE6
98 LINES	FDAD	99 CLS	FD8B	9A WIDTH	FDA5	9B MONITOR	FEA2
9C SET	1054	9D RESET	1057	9E PRINT	EB23	9F CONT	E89E
A0 LIST	E6DD	A1 CLEAR	E9CA	A2 CLOAD	F4F9	A3 CSAVE	F4C3
A4 NEW	E4B9	A5 TAB(	EBAF	A6 TO		A7 FN	F133
A8 SPC(	EBAF	A9 THEN		AA NOT	EF08	AB STEP	
AC +	F994	AD -	F5C8	AE *	F706	AF /	F767
B0 ^	FAB5	B1 AND	EE81	B2 OR	EE80	B3 >	
B4 =		B5 <		B6 SGN	F822	B7 INT	F8E6
B8 ABS	F838	B9 USR	1003	BA FRE	F0D0	BB INP	F441
BC POS	F0FE	BD SQR	FAAC	BE RND	F88B	BF LOG	F6C7
C0 EXP	FAFA	C1 COS	FC00	C2 SIN	FC06	C3 TAN	FC67
C4 ATN	FC7C	C5 PEEK	F5A3	C6 DEEK	FDBC	C7 POINT	1051
C8 LEN	F382	C9 STR$	F19A	CA VAL	F41C	CB ASC	F391
CC CHR$	F3A2	CD LEFT$	F3B2	CE RIGHT$	F3E2	CF MID$	F3EC

APPENDIX B

And now some tips on using the info in your own machine code routines. If you wish to return to Basic signalling an error JP to one of the following locations. Note that some are conditional to the state of one of the flags in the F register which must be set accordingly before the JP, all error signals are available, those listed are the ones most likely to be useful (even if they are not used in the correct context).

ERROR	JP ADDRESS	CONDITION
SN	E3AD	
OD	ECDD	Z
FC	E9A0	
OV	E3BC	
OM	E3A2	
UL	EA46	
BS	F045	
TM	E3BF	
OS	F248	Z
LS	F31A	C
MO	EDD8	Z

By far the most useful facility that I have found is to be able to supply additional information to a user program by adding extra information within a

Basic line after the USR(X) statement. Just a few ideas.... pass string/numeric array name to disc array saving routine, start and increment line numbers to re-number routine, file name to a named tape or disc file saving routine, etc. etc. etc. In order to get at this info we have to get the text pointer from where ever Basic stored it before we came it came to your routine, and then having picked off our data increment the pointer passed this data so that an SN error does not occur on our return to Basic. (Basic will be expecting it to point at a colon or a line terminator). The text pointer is to be found above the Basic return address on the stack and it points to the first non-space character after the USR(X) statement. A typical assembly listing to obtain it follows . in this example start and increment values are passed to a re-number routine in the format U=USR(0)10,5

```
        LD HL £0A       ;set default start line and increment
        LD (START) HL   ;to 10
        LD (INCR)  HL   ;
        POP HL          ;get text pointer
        EX (SP) HL
        CALL £E837+1    ;test if 1st character is ASCII decimal
                        ;entry at E838 rather than E837 so that
                        ;HL is not incremented passed 1st char.
                        ;on entry
        JR NC RENUM     ;NC so no number, use default values
        CALL £E9A5      ;convert ASCII number to hex in DE
        LD (START) DE   ;save start number
        LD A (HL)       ;is there a comma following the
        CP £2C          ;number
        JR NZ RENUM     ;if no use default increment
        CALL £E9A5+1    ;convert increment ASCII number
        LD (INCR) DE    ;save increment value
RENUM   EX (SP) HL      ;restore text pointer
        PUSH HL
```

With a re-number routine restoration of the text pointer is not in fact necessary as Basic has to be re-initialised on return as the program length will most likely have been changed by re-numbering, but more on that later.

The re-number routine then searches the Basic program looking for GOTOs GOSUBs etc and CALLs can then be made to £E9A5 to convert the ASCII strings following them to a hex line number in DE (Note that this routine like many many routines in Basic returns with the text pointer pointing at the first non-space character after that part of the Basic line which it has used). A search is then made for the line number and the new value after re-numbering calculated. In order to convert this hex number to an ASCII string load it into HL and CALL F9AD. The ASCII string will be found in the string buffer at 104A and is terminated by a 0 (the number will also be printed on the screen). typical assembly listing :-

```
        LD HL LINNUM    ;load new line number
        CALL £F9AD      ;convert it to ASCII string
        LD HL £10EA     ;point to string in buffer
        LD DE (TEXTP)   ;load DE with stored text pointer
NXTASC  LD A (HL)       ;load first ASCII character
        OR A            ;test for end of string
        JR Z NUMDON     ;finish if end of string
        LD (DE) A       ;insert new number in Basic line
        INC DE          ;increment text and
        INC HL          ;string pointers
        JR NXTASC       ;get next character
NUMDON                  ;re-number prog continues
```

This is of course a very incomplete listing as checks to ensure wether the new number is a different size to the old, if there is sufficient memory space to insert it etc. would be required.

Any routine, like a re-number routine, which can alter the size of a Basic program will have messed up the start of line pointers and the top of program/variable pointers in the Basic workspace, the following solves these problems :-

```
        LD HL £10FA       ;point HL to start of Basic prog
READDR  PUSH HL           ;save prog pointer
        LD E (HL)         ;test start of next line pointer
        INC HL            ;if 0 then top of prog
        LD D (HL)         ;
        INC HL            ;if top of prog HL will point to new
        LD A E            ;top for workspace initialising
        OR D              ;
        POP DE            ;recover prog pointer
        JR Z PTOP         ;finish re-addr if top of prog
        INC HL            ;bump passed line number
        INC HL            ;
FNDEND  LD A (HL)         ;find end of Basic line
        INC HL            ;
        OR A              ;
        JR NZ FNDEND      ;
        LD A L            ;insert new next line pointer
        LD (DE) A         ;
        LD A H            ;
        INC DE            ;
        LD (DE) A         ;
        JR READDR         ;do next line
PTOP    CALL £E4C2        ;re-initialise workspace
        JP £E3F8          ;return to Basic
```

As stated earlier a routine which may expand a Basic program should make sure that it is not expanded too far. With a re-number routine reaching the limit would be disasterous as the Basic program would be only partially re-numbered and therefore corrupt, but with a routine which, for instance, inserts spaces to make "pretty" listings no harm will have been done and if the limit has been reached sufficient room should have been left so that a USR call may be made to remove the spaces. I have found that 52H bytes must be left below the top of stack. The listing below shows an example, on entry HL would contain the new planned program top.

```
        PUSH HL           ;save planned top
        LD DE (£105A)     ;get stack top
        LD HL -£52        ;calculate limit
        ADD HL DE         ;
        POP DE            ;recover planned top
        CALL £E68A        ;compare DE with HL
        JP C £E3A2        ;if C then passed limit so return to
                          ;Basic flagging OM Error
```

Depending on what context this routine is being used with it may well be necessary to re-address the start of line pointers and re-initialise Basic before returning.

The ordinary USR statement can only pass 1 value to a user program, but by inserting variable names after the USR statement it is possible to pass numeric values or strings under Basic program control. a routine at EF2D will find the variable data concerned.

```
        POP IX          ;save return address
        POP HL          ;get text pointer
        CALL £EF2D      ;find variable accumulator
                        ;routine returns with DE pointing to
                        ;the first byte of the accumulator
        PUSH HL         ;replace text pointer
        PUSH IX         ;restore return address to stack
        LD HL £10E4     ;point HL to workspac accumulator
        EX DE HL        ;move variable accumulator to
        CALL £F86B      ;workspace accumulator
        CALL £E98B      ;convert to a hex number in DE
```

Note that in the application above the routine at E98B (the routine used to obtain the USR argument) has been used, so only an accumulator in the valid range should be used (see notes).

If you want the variable find routine at £EF2D to find and point to the start of a numeric or string array, not just at an element within it (e.g for a routine to store numeric or string arrays on disc file) the contents of 10CB in the workspace should be set to 01.

If you wish to run a Basic program called by a machine code program then PUSH E7F2 onto the stack and JP to E4C5.

I've not tested the following but it seems reasonable to be able to RUN or GOTO a specific line number by loading DE with required line number as a hex value and JPing to EA3F. For example to RUN a Basic program from line 10:-

```
        LD HL RETURN    ;set USR vector for return
        LD (£1004) HL   ;from Basic prog
        CALL £E4C9      ;initialise variables (RUN only)
        LD HL £E7F2     ;put false RET on stack
        LD DE £000A     ;load DE with line 10
        OR A            ;reset carry flag
        JP EA3F         ;JP to Basic
RETURN                  ;user prog continues
```

Well I hope these ideas provide food for thought, there must be many more things to do and I would be interested to hear from anybody with either queries or news of their applications. A.S.Watkins, 7, Warwick Close, Maidenhead, Berkshire, SL6 3AL.